Ассоциативные контейнеры

```
set<T>, map<K,V>, unordered_set<T>, unordered_map<K,V>
multiset<T>, multimap<K,V>
```

```
set<int> s;
auto pr = s.insert(3); // возвращается pair(it,b)
auto it = s.find(3); // it == s.end()
bool b = s.erase(3);
s.count()
for (auto x : s) // цикл по set
  cout<<x<<" ";
```

```
map<string,int> m;
v=m[k]  если нет k, то (k, V()) в конец
auto it = m.find(k) // it – это pair(K,V)
bool b = m.erase(k);
m.count()
Цикл:
for (auto &x : m)
  cout<<x.first<<" "<<x.second<<endl;
for (auto it=m.begin(); it!=m.end(); ++it)
  cout<<(*it).first<<" "<<it->second<<endl;
```

```
set<Person> sp;
либо в Person д.б. определена операция <
либо set<Person,PersonLess> sp; где
struct PersonLess {
  bool operator() (const Person& p1,
   const Person& p2) const { … }
};
либо
auto cmp = [](const Person& p1, const Person& p2)
  { return p1.name < p2.name; };
set <Person, decltype(cmp)> sss(cmp);
```

Эквивалентность вместо равенства: (!)

```
Equiv(a,b) == !cmp(a,b) && ! cmp(b,a)
```

```
map<Key, Value> m;
  Для Key – operator< или cmp (равенство определяется как эквивалентность)
  Для Value – K. по умолчанию
```

```
map<Person,int> m;
```

```
либо в Person д.б. определена операция <
либо map<Person,int,PersonLess> m;
либо map <Person,int,decltype(cmp)> mmm(cmp);
```

```
unordered_set<K, Hash=std::hash<K>(), Eq=std::equal_to<K>()>
unordered_set<Person,Hash,Eq> s;
Hash,Eq – классы с operator():
struct Hash {
size_t operator()(const Person& p) const {
    return hash<string>()(r.name) ^ hash<int>()(r.age);
  }
};
struct Eq {
  size_t operator()(const Person& p1,
    const Person& p2) const {
      return p1.name==p2.name && p1.age == p2.age;
    }
};
```

```
либо
auto hf = [](const Person& r)
  { return hash<string>()(r.name)^hash<int>()(r.age); };

auto eq = [](const Person & r, const Person & r2)
  { return r.name==r2.name && r.age==r2.age; };

unordered_set<Record,decltype(hf),decltype(eq)> m {10,hf,eq};
```

либо

```
namespace std {
  template<>
  struct hash<Person>
  {
    size_t operator()(const Person &r) const
    { return hash<string>()(r.name)^hash<int>()(r.age); }
  };
  template<>
  struct equal_to<Person>
  {
    bool operator()(const Person & r, const Person & r2) const
    { return r.name==r2.name && r.age==r2.age; }
  };
}
unordered_set<Person> m;
```

Умные указатели

unique_ptr: единоличное владение объектом через его указатель; разрушает объект, когда unique_ptr выходит из области видимости; безопасен к исключениям т.к. механизм обработки И. вызывает деструкторы локальных объектов. unique_ptr нельзя копировать и присваивать!

```
unique_ptr<int> pp1(new int(5));
cout << *pp1 << endl;
auto pp = make_unique<int>(101);
cout << *pp << endl;
unique_ptr<Person> p(new Person("Ivanov", 20));
p = make_unique<Person>("Petrov", 19);
cout << *p << p->name();
// форма для массива
unique_ptr<int[]> pi(new int[10]);
pi[0] = 777;
cout << pi[0] << endl;
cout << (bool)pi << endl;
int* ii = pi.release();
cout << (bool)pi << endl;
```

shared_ptr: умный указатель, с владением объектом через его указатель с использованием механизма подсчёта ссылок. Несколько указателей shared_ptr могут владеть одним и тем же объектом; объект будет уничтожен, когда последний shared_ptr, указывающий на него, будет уничтожен или сброшен

```
shared_ptr<Person> sp(new Person("Petrov",21));
shared_ptr<Person> sp1 = sp;
auto sp2 =make_shared<Person>("Kozlov",23);
cout << sp2.use_count() << endl;
sp2 = sp;
cout << sp2.use_count() << endl;
cout << *sp;
sp.reset(new Person("Oslova",20));
```

weak_ptr: ссылается на объект, которым владеет shared_ptr. Не увеличивает счетчик ссылок. Может в любой момент быть зафиксирован и преобразован в shared_ptr.

```
weak_ptr<Person> wp;
{
    auto sp = make_shared<Person>("Kozlov", 23);
    wp = sp;
    cout << sp.use_count() << endl;
    cout << wp.expired() << endl; // 0
}
cout << wp.expired() << endl; // 1
…
if (auto q = wp.lock()) // q – shared_ptr
  …
else …
```